

Computer Science Department

TECHNICAL REPORT

PARALLEL COMPUTATION ON 2-3 TREES

BY

W. Paul, U. Vishkin and H. Wagener

Technical Report - #70

Ultracomputer Note #52

April 1983

NEW YORK UNIVERSITY



Department of Computer Science
Courant Institute of Mathematical Sciences

251 MERCER STREET, NEW YORK, N.Y. 10012

CS TR. 70
C. 2



PARALLEL COMPUTATION ON 2-3 TREES

BY

W. Paul, U. Vishkin and H. Wagener

Technical Report - #70

Ultracomputer Note #52

April 1983

New York University
Department of Computer Science
Courant Institute of Mathematical Sciences
251 Mercer Street
New York, N. Y. 10012

PARALLEL COMPUTATION ON 2-3 TREES*

W. Paul**

IBM Research Laboratory
San Jose, California 95193

U. Vishkin

Courant Institute of Mathematical Sciences
New York University, 251 Mercer Street
New York, New York 10012

and

H. Wagerer

Technische Universitaet Berlin
Institut fuer Software and Theoretische Informatik
Strasse des 17. Juni 135, D-1000 Berlin 10, West Germany

ABSTRACT

Our model of computation is a parallel computer with k synchronized processors $P_1, \dots, P_k$ sharing a common random access storage, where simultaneous access to the same storage location by two or more processors is not allowed. Suppose a 2-3 tree T with n leaves is implemented in the storage, suppose $a_1, \dots, a_k$ are data that may or may not be stored in the leaves, and for all i , $1 \leq i \leq k$, processor P_i knows a_i . We show how to search for $a_1, \dots, a_k$ in the tree T , insert these data into the tree, delete them from the tree, split the tree with respect to these elements and perform the union of $k+1$ range-disjoint 2-3 trees in $O(\log n + \log k)$ steps.

1. Introduction

Technology will make it possible to build computers with a large number of cooperating processors in the near future. However, building such computers will only be worthwhile if the increased computing power can be used to reduce considerably the execution time of sufficiently many basic computational problems. In particular, one would like to have datastructures, where k processors can solve many problems about k times faster than a single processor. 2-3 trees are one such datastructure as

*A preliminary version of this paper will be presented at the Tenth ICALP, Barcelona, July 1983. **Part of this research was done while the first

author was visiting the Institut de Programmation of the Université Paris VI.

will be demonstrated here. Protocols that avoid read or write conflicts, if several processors are working simultaneously on the same balanced tree, have been studied previously [BS], [S], but apparently no attempt was made to design fast algorithms and to analyze their running time.

In the sequel, we say very little about how to avoid read or write conflicts. In the situations where they are possible, there are easy ways to avoid them. We will, however, have to say some words about storage allocation.

2. 2-3 Trees

A 2-3 tree T is a tree in which all leaves have the same depth and each interior node v has two or three sons: the left son $l(v)$, the right son $r(v)$, and in case there are three sons, the middle son $m(v)$. Data from a totally ordered domain are stored in the leaves with smaller data to the left of larger ones. For each node v , the value $L(v)$ (resp. $R(v)$) of the largest element stored in the subtree of T with root $l(v)$ (resp. $r(v)$) is stored in v . Recall that in the sequential use of 2-3 trees $R(v)$ is not stored. If v has three sons, then the value $M(v)$ of the largest element stored in the subtree of T with root $m(v)$ is also stored in v . The depth of a node v in T is its distance from the root, the height of v is its distance from the leaves. We assume the reader to be familiar with the usual search, insertion, deletion and split routines as described say in [AHU].

Suppose a 2-3 tree T with n leaves is implemented in the storage, suppose $a_1, \dots, a_k$ are data that may or may not be stored in the leaves, suppose $a_1 < \dots < a_k$ and for all i processor P_i knows a_i . We show how to perform any of these four operations with respect to the k elements by the k processors in $O(\log n + \log k)$ steps. Say that $T_0, T_1, \dots, T_k$ are $k+1$ 2-3 trees such that the elements which are stored in their leaves are taken from pairwise disjoint intervals. We show how to perform the union of these trees into a 2-3 tree in $O(\log n + \log k)$ steps.

If the elements (or the trees for the union operation) $a_1, \dots, a_k$ arrive unsorted they can be sorted in $O(\log k)$ time (see [AKS]). Their solution can readily be modified into our model of computation using k processors (see [V2] for similar arguments).

3. Search

If simultaneous access by several processors to the same storage location for read purposes is allowed (as in the PRAM model of parallel computation of [FW]) then search is very simple. Processor P_i ($1 \leq i \leq k$) performs the standard sequential search of a_i in $O(\log n)$ time ([AHU]). Since no writes into the shared memory are required, this is done in parallel by all k processors in time $O(\log n)$. Since such simultaneous reads are not allowed in this presentation, we need another solution.

A chain is a subsequence $a_f, a_{f+1}, \dots, a_\ell$ of the input sequence $a_1, \dots, a_k$. Such a chain corresponds in a natural way to a chain of processors $P_f, P_{f+1}, \dots, P_\ell$. The search algorithm starts with the chain $a_1, \dots, a_k$ at the root of the 2-3 tree T . This chain is subsequently split into many subchains which are wandering down the tree. Among the processors of a chain $a_f, \dots, a_\ell$ only the first one, i.e., P_f , is active. P_f knows ℓ and of course f . If at some time the chain is split into $a_f, \dots, a_{m-1}$ and $a_m, \dots, a_\ell$, then processor P_f will invoke processor P_m and transmit the value ℓ to P_m .

The search algorithm proceeds in stages. During each stage s , the active processor of each chain C will access the data in some node v of the 2-3 tree T . We say that C is in node v at stage s . The chain $a_1, \dots, a_k$ is in the root at stage 1. During each stage, each active processor processes its chain once. We describe how this is done.

Suppose a chain $C = a_f, \dots, a_\ell$ is in node v at stage s , the node v has two or three sons and the labels $L(v)$ and possibly $M(v)$ are stored in v . We say that C hits a label X , if $a_f \leq X < a_\ell$. The label $R(v)$ does not play any role in the present discussion.

Chains C that hit no label are sent to the appropriate son of v ; more precisely: C is at stage $s+1$ in node

$$\ell(v) \quad \text{if } a_\ell \leq L(v)$$

$$m(v) \quad \text{if } L(v) < a_f \text{ and } a_\ell \leq M(v) \text{ and } v \text{ has 3 sons}$$

$$\begin{aligned} r(v) \quad & \text{if } M(v) < a_f \text{ and } v \text{ has 3 sons} \\ & \text{or } L(v) < a_f \text{ and } v \text{ has 2 sons.} \end{aligned}$$

For chains $C = a_f, \dots, a_\ell$ let $C_1 = a_f, \dots, a_{m-1}$ and $C_2 = a_m, \dots, a_\ell$ with $m = \lceil (f+\ell)/2 \rceil$. If C hits a label, then it is split into C_1 and C_2 . If C_i , $i = 1, 2$, hits no label, then it is sent to the appropriate son, else

it remains in v , i.e., C_1 is in node v at stage $s+1$. Clearly, a chain can be processed in $O(1)$ steps.

Claim. (a) (resp. (b)). Say that elements $a_f, a_{f+1}, \dots, a_\ell$ only have passed through edge e of T through stage s , for any $s \geq 1$. If a chain C such that $a_j \in C$ and $j > \ell$ (resp. $j < f$) passed through e at stage $s+1$ then $a_{\ell+1} \in C$ (resp. $a_{f-1} \in C$).

Corollary. No more than two chains may pass each edge e of T at any single stage.

Proof of Claim. By induction on the depth of e in T . The claim obviously holds for each edge e that emanates from the root of T . This completes the base of the induction. Assume that both parts (a) and (b) of the claim hold for all edges of depth k . Let $e_1 = (v, w)$ be an edge of depth $k+1$ and $e_2 = (u, v)$ its father edge in T . We will show that part (a) of the claim holds for e . The proof of part (b) is similar.

Elements $a_f, a_{f+1}, \dots, a_\ell$ (resp. a_j) passed e_2 through stage $s-1$ (resp. s). By the inductive hypothesis $a_{\ell+1}$ passed e_2 no later than stage s .

Case 1. w is a left son of v .

If $a_{\ell+1}$ passed e_2 at stage s then, by the inductive hypothesis, $a_{\ell+1}$ passed e_2 at the same chain as a_j . If $a_{\ell+1}$ passed e_2 before stage s then $a_{\ell+1}$ and a_j were, again, in the same chain at e_2 since otherwise $a_{\ell+1}$ would not have been delayed at v . Since left chops of hit chains are sent to left sons and $a_{\ell+1}$ did not pass e_1 before stage $s+1$, $a_{\ell+1}$ and a_j pass e_1 in the same chain at stage $s+1$.

Case 2. w is a right son of v .

The chain in which a_ℓ passed e_2 did not contain $a_{\ell+1}$ because if it did then $a_{\ell+1}$ would have passed e_1 not later than a_ℓ . So, since a_j (and $a_{\ell+1}$) could not have delayed at v it passed e_2 at stage s and by the inductive hypothesis its chain included $a_{\ell+1}$. This chain passed e_1 at stage $s+1$.

Case 3. w is a middle son of v .

If the chain in which a_ℓ passed e_2 contained $a_{\ell+1}$ then it must have contained a_j and large enough elements to hit label $M(v)$; now, if it did not hit $L(v)$ then the left-chopping arguments (see Case 1) imply that $a_{\ell+1}$ and a_j passed e_1 in the same chain. If it hit $L(v)$ then this chain (or later subchains of it) are cut into pieces that separate a_ℓ and a_j ; $a_{\ell+1}$ must be in the right one with a_j (otherwise it is sent on e_1 no later than a_ℓ) and, again, the left-chopping arguments apply. If the chain in which

a_ℓ passed e_2 did not contain $a_{\ell+1}$ then the analysis of Case 1 applies. This completes the proof of the claim.

The corollary implies that for each s and v at most 4 chains are in v at stage s . Thus, each stage lasts $O(1)$ steps. Once a chain $a_f, \dots, a_\ell$ has arrived in a leaf b , the processors $P_{f+1}, \dots, P_\ell$ have to be informed of the value of b . This is done recursively in $\log k$ stages. In stage j , $0 < j < \lceil \log k \rceil - 1$, processor P_i that knows where a_i falls informs P_{i+2^j} , if this later processor does not know yet where a_{i+2^j} falls. See [VI] for more details. Whenever a chain hits a label it is halved. Thus, any element may be contained in chains that hit labels no more than $\lceil \log k \rceil$ times; therefore it arrives to a leaf in at most $\lceil \log n \rceil + \lceil \log k \rceil$ stages, and the search takes $O(\log n + \log k)$ time.

4. Insertions

The tree T has n leaves $b_1 < b_2 < \dots < b_n$. The elements $a_1, \dots, a_k$ are to be inserted into T .

We first run the search algorithm. This results in splitting the input into chains (a_i and a_j , $1 < i < j < k$, belong to the same chain if there is no leaf b_q , $1 < q < n$, such that $a_i < b_q < a_j$). There are $n+1$ possible chains $C_0, C_1, \dots, C_n$. Let $|C_q|$ denote the number of input elements in chain C_q . (For most q , $0 < q < n$, $|C_q| = 0$ since it makes sense to insert elements to a tree rather than building it from scratch only if $k \ll n$.) We say that by the insertion algorithm elements of chain C_q (resp. C_0) arrive to leaf b_q (resp. b_1) and fall to its right (resp. left) hand side, for $1 < q < n$.

First, we describe a simple algorithm for the special case $|C_0| = 0$ and $|C_q| < 1$ for all $1 < q < n$. This algorithm works in stages. In stage 1, for all i processor P_i makes a_i a son of a father of b_1 and then stands by on a_i . Now the algorithm works such that for all s after stage s the following holds:

All leaves in the tree have the same depth, all interior nodes of height $\neq s$ have two or three sons. Between each pair of "old" nodes (resp. to the right of the rightmost old node) of height $s-1$, there is at most one "new" node of height $s-1$. Each such new node has a processor standing by. So an (old) node v of height s has at most three new sons. It also has no more than three old sons. In an obvious way the processor standing by at a leftmost new son of a node of size s "takes over," while

the other two processors become inactive. In case the total number of old and new sons of v is ≤ 3 the new son becomes an "ordinary" son of v and the processor becomes inactive. In case this total number of sons is > 3 , a new internal node v' of height s which becomes the right brother of v is created and the new and old sons of height $s-1$ are partitioned properly among v and v' . The processor then stands by on v' . Updating the L , M and R fields of v and v' in both cases is easy. So stage s takes $O(1)$ steps. We showed that in each stage, several new nodes of the tree may be created simultaneously. We will say later how to do this without occupying too much storage space.

Let us go back to the general problem of insertion. If $|C_0| > 0$, start by inserting a_1 by the sequential algorithm. The new C_0 (with respect to the new tree and $a_2, a_3, \dots, a_k$) satisfies $|C_0| = 0$.

The problem of inserting a long chain $C_j = a_f, \dots, a_k$ at leaf b_j , for $1 \leq j \leq n$, is reduced to the problem of inserting shorter chains. This is done by first inserting the middle element a_m ($m = (f+k)/2$) at leaf b_j and then inserting recursively $a_f, \dots, a_{m-1}$ at b_j and $a_{m+1}, \dots, a_k$ at a_m . This is done for all chains in parallel and the middle elements are inserted by the simple algorithm described above. After the chains have been split $\log k$ times, they are reduced to length one. Thus, running first the algorithm for C_0 and then the simple algorithm $\lceil \log k \rceil$ times would do the job in $O(\log n \log k)$ steps. For $i \leq \lceil \log k \rceil$, let T_i be the tree obtained by running the algorithm for C_0 and then the simple algorithm i times. Now for all i running the simple algorithm the i 'th time results in a wave of processors running up T_{i-1} at a speed of one level per stage, and below this wave, the tree already looks like T_i . Thus, pipelining can be applied; this is since before starting the $(i+1)$ -st run of the simple algorithm and with it the $(i+1)$ -st wave of processors, one has not to wait until the i 'th wave has reached the root, but only long enough to ensure that the two waves will not overlap. Three stages will certainly suffice.

5. Deletions

For deleting the elements $a_1 \leq a_2 \leq \dots \leq a_k$ from the 2-3 tree T , we first run the search algorithm. Similar to the description of the insertion algorithm we have elements of chain C_q (resp. C_0) which arrive to leaf b_q (resp. b_1) and fall to its right (resp. left) hand side, for

$1 < q < n$. C_0 cannot contain an element which is in a leaf and therefore can be ignored. For each (non-empty) chain C_q ($1 < q < n$) the processor of its left-most element checks if its element is the same as the one stored in leaf b_q . If it is so, the processor marks leaf b_q for deletion and stands by on this leaf. All other processors become inactive.

First, we describe a simple algorithm for the special case where each node of height 1 has at least one non-marked son. This algorithm works in stages.

In stage 1 all the marked leaves are deleted. Following stage 1 a node of height 1 may either remain a node of height 1 or become a node of height 0 (a leaf). Say that a node v of height 1 had one or two son-leaves which were deleted. If it remained of height 1 (it is possible if it had three sons and only one was deleted) then the processor of this deleted leaf becomes inactive. Otherwise the processor of a deleted son-leaf stands by on v and v is marked. The processor of a (possibly existing) second deleted son-leaf becomes inactive.

The algorithm works such that, for all s ($s > 1$), the following holds after stage s :

1. Each marked node is a root of a 2-3 tree of height $s-1$ and is a son of a node of height $s+1$ in T . It has a processor standing by on it.
2. Each internal node has two or three sons.
3. Each node of height s in T which is not marked is a root of a 2-3 tree of height s .

It is easy to verify that each node v of height $s+1$ in T which has a marked son must have between two and seven sons and grandchildren of height $s-1$. All, but one, of the processors which stand by on a marked son of v become inactive. This processor does the following in stage $s+1$:

- if v has > 4 sons and grandchildren of height $s-1$ then they are partitioned in the usual way into sons of v so as to make v of height $s+1$ as before. The processor becomes inactive.
- else the nodes of height $s-1$ become sons of v_1 in the usual way, v is marked and the processor is standing by on v .

Care has to be taken in order to avoid, read or write conflicts and to choose at each stage the processors which become inactive. This as well as updating the L , M and R fields is easy. The algorithm runs in $O(\log n)$ time.

Let us go back to the general case. We run the following algorithm

A. It works in stages. Denote our 2-3 tree T by T_0 . Let T_t be the 2-3 tree which is the output of stage t for $t \geq 1$.

Stage t ($t \geq 1$)

T_{t-1} is the input 2-3 tree for stage t . For each node of height 1 in T_{t-1} such that all its son-leaves have to be deleted mark all leaves, but one, for deletion. For each node such that not all its son-leaves have to be deleted mark the ones that have to be deleted. Processors of these marked leaves stand by on them. Processor of leaves that have to be deleted but have not yet been marked do not take part in the rest of this stage. The stage proceeds, now, in the same way as the algorithm for the special case given above.

In each stage we delete at least half of the leaves that have to be deleted but have not been deleted by previous stages. Therefore, algorithm A runs in $\leq \log k$ stages. Similar to Section 4 (insertions) we pipeline the stages of algorithm A thereby obtaining an overall time complexity of $O(\log n + \log k)$.

6. Splits

Given k elements $a_1 < a_2 < \dots < a_k$ we would like to split the tree T with respect to them. Namely, if the leaves of T represent a set of elements S then the output of the split operation is $k+1$ 2-3 trees $T_0, T_1, \dots, T_k$ such that the leaves of tree T_i represent the elements of S_i ; where, $S_i = \{a \in S: a_i < a < a_{i+1}\}$ for $1 \leq i \leq k-1$, $S_0 = \{a \in S: a < a_1\}$ and $S_k = \{a \in S: a > a_k\}$.

In order to clarify the presentation, let us start with a solution for the corresponding sequential split problem, i.e., we give an algorithm which employs a single processor for the splitting of a 2-3 tree with respect to a single element.

1. Using the standard search algorithm, find the path $\pi(a) = (p_1, \dots, p_t)$ from the root of T to a . Delete the whole path $\pi(a)$ and all edges adjacent to it from T .

One is left with a forest of subtrees of T some of which were to the left of the path $\pi(a)$ and the others were to the right of $\pi(a)$. Let us call these subtrees of T the left (resp. right) side trees of path $\pi(a)$. If a was not stored in the tree, then a was eventually compared to a leaf b of T with the result $a < b$ or $a > b$. In the first [second] case treat b , and possibly its right [left] brothers, as right [left] side trees of $\pi(a)$.

2. Join the left (resp. right) subtrees of $\pi(a)$ into a 2-3 tree L (resp. R).

We describe this step for the left subtrees only. Right subtrees are handled similarly. This is done in stages which are described recursively. Before starting stage s we have a 2-3 tree S that contains the leaves of all left subtrees of height $< s-1$. The height of S is $h < s$. There are either none or one or two left subtrees of height s . In the second (resp. third) case we denote them L (resp. L_1 and L_2). In the first case S is the "output" of stage s . In the third case L_1 and L_2 are joined into a 2-3 tree of height $s+1$ in the obvious way. Let us denote this new tree by L , too. No confusion will arise. In both the second and third cases we reach, down the rightmost path in L , a node of height $h+1$. The root of S becomes its son and the insertion is propagated in the standard way up in L . (The possibility where S is of height s and we are in the second case is simple and has to be added to the description.) This completes the description of one step. Note that the height of the new S is $< s+1$.

Throughout step 2 we did not visit any level more $O(1)$ time; in stage s we never go below the height h of S before the stage started. Each time we visit a level $O(1)$ operations are performed. Therefore, Step 2 takes $O(\log n)$ time.

We now parallelize this algorithm in order to split simultaneously a 2-3 tree T with respect to elements $a_1, \dots, a_k$:

1. Run the search algorithm for $a_1, \dots, a_k$ and mark the paths $\pi(a_1), \dots, \pi(a_k)$. As we do not require the elements a_i to be stored in the tree T , these paths are not necessarily distinct.

For all i , we define the left [right] forest $LF(i)$ [$RF(i)$] of path $\pi(a_i)$ as the set of left [right] side trees of $\pi(a_i)$ whose root is not marked and that are not left [right] side trees of $\pi(a_{i-1})$ [$\pi(a_{i+1})$]. The example of Fig. 1 shows the only case where $RF(i)$ and $LF(i+1)$ may have a tree in common. "Correct" our definition for this case, so that this tree belongs to $LF(i+1)$ only.

2. Rerun the search algorithm, but for all chains $a_f, \dots, a_\ell$ that are created have the processors p_f and p_ℓ both active. Processor $P_f[p_\ell]$ keeps track which trees are in $LF(f)$ [$RF(\ell)$]. Also delete in this run the paths $\pi(a_1), \dots, \pi(a_k)$ and the adjacent edges. For all chains

$a_f \dots a_\ell$ that have reached a leaf only processors P_f and P_ℓ remain active. P_f remembers the index ℓ of the next active processor. The following two commands are executed by active processors P_i only, similar to Step 2 of the sequential algorithm.

3. For all P_f processors: join the left forest of path $\pi(a_f)$ into a 2-3 tree L_{f-1} .
4. For all P_ℓ processors: join the right forest of path $\pi(a_\ell)$ into a 2-3 tree R_ℓ . (All R_i and L_i that were not affected by the last two commands are empty.)
5. (a) Processor P_1 : Insert L_0 into T_0 .
Processor P_k : Insert R_k into T_k .

(b) For all P_i processors ($1 \leq i < k$): Join L_i and R_i into a 2-3 tree T_i . (The "insertion" into the T_i trees should be understood as renaming rather than copying.)

The split algorithm takes $O(\log n + \log k)$ time.

Note that throughout this section we omitted the updates of the fields $L(v)$, $M(v)$ and $R(v)$. It is always easy to complete these details.

7. Unions

Let $T_0, T_1, \dots, T_k$ be 2-3 trees such that the elements that are represented by their leaves satisfy the following: if a leaf of T_i (resp. T_j) is represented by element a_i (resp. a_j) and $0 \leq i < j \leq k$ then $a_i < a_j$. Our problem is to join $T_0, T_1, \dots, T_k$ into a new 2-3 tree. As was done before, we start with a solution of the corresponding sequential union problem: Join T_0 and T_1 into a 2-3 tree.

A fairly awkward way of performing this is given. However, this technique is useful in the parallel algorithm since it enables pipelining. Start at the rightmost leaf of T_0 and the leftmost leaf of T_1 . Climb, level by level, simultaneously in T_0 and T_1 , till the first root of either T_0 or T_1 is encountered. Then join T_0 and T_1 to one tree and propagate the update, in the standard way, to form a 2-3 tree. The algorithm requires $O(\log n)$ time. See [AHU] for all "standard ways" mentioned above.

We now parallelize and pipeline this algorithm in order to compute the union of $T_0, T_1, \dots, T_k$. Let us denote them by $T_0^{(o)}, T_1^{(o)}, \dots, T_k^{(o)}$, respectively. This is done in phases $j = 0, 1, \dots$.

For any tree let $\lambda(T)$ (resp. $\rho(T)$) denote the leftmost (resp. rightmost) leaf of T . For all odd i , processor P_i determines: $\lambda(T_{i-1}^{(o)})$, $\rho(T_{i-1}^{(o)})$, $\lambda(T_i^{(o)})$ and $\rho(T_i^{(o)})$.

The following is true for $j = 0$ and will remain true: At the beginning of phase j we are left with $(k+1)/2^j$ trees $T^{(j)}, T^{(j)}, \dots$ where for each $v \in \{0, \dots, (k+1)/2^j - 1\}$ the tree $T_v^{(j)}$ is a 2-3 tree obtained by joining $T_{v2^j}^{(j-1)}, \dots, T_{(v+1)2^j-1}^{(j-1)}$. For each v we have not yet used processor P_{v2^j} and processor P_{v2^j} knows $\rho(T_{v-1}^{(j)})$, $\lambda(T_{v-1}^{(j)})$, $\rho(T_v^{(j)})$ and $\lambda(T_v^{(j)})$.

The following is done in phase j : For each odd v processor P_{v2^j} runs up the right branch of $T_{v-1}^{(j)}$ and the left branch of $T_v^{(j)}$ and joins the two trees into $T_{(v-1)/2}^{(j+1)}$, as in Step 3 of the sequential algorithm. This processor performs also,

$$\lambda(T_{(v-1)/2}^{(j+1)}) + \begin{cases} \lambda(T_{v-1}^{(j)}) & \text{if } T_{v-1}^{(j)} \neq \phi \\ \lambda(T_v^{(j)}) & \text{otherwise.} \end{cases}$$

and

$$\rho(T_{(v-1)/2}^{(j+1)}) + \begin{cases} \rho(T_v^{(j)}) & \text{if } T_v^{(j)} \neq \phi \\ \rho(T_{v-1}^{(j)}) & \text{otherwise.} \end{cases}$$

Finally, observe that the phases can be pipelined, i.e., for all j , phase $j+1$ can be started a constant number of steps after phase j . The union algorithm takes $O(\log n + \log k)$ time.

Note that throughout this section we omitted the updates of the fields $L(v)$, $M(v)$ and $R(v)$. It is always easy to complete these details.

It is interesting to note that performing first the split algorithm and then the union algorithm yields an alternative deletion algorithm.

8. Storage Allocation

Nodes of a 2-3 tree with n leaves are stored in the first, say N , locations of some vector A . During the insertion algorithm, each processor P_i may create $n_i < \log n$ new nodes. Therefore, for each processor P_i $\log n$ consecutive locations of some other vector B are reserved, where the new nodes are created. For each i let $N_i = \sum_{j < i} n_j$. After the insertion algorithm, the numbers N_i are computed in parallel (in $O(\log k)$ time) and for all i processor P_i copies the nodes that it created into rows $N+N_i+1, \dots, N+N_i+n_i$ of A .

During command 2 of the split algorithm, each processor P_i may cancel $m_i < \log n$ nodes, i.e., locations in A . Each processor P_i stores the numbers of these rows in its private section in the vector B . After command 2 of the split algorithm $M = \sum_{i=1}^k m_i$ is computed. Now the rows with numbers $> N-M$ that were not cancelled have to be copied into the rows with numbers $< N-M$, that have been cancelled: locations $N-M+1, \dots, N$ of A are partitioned into blocks $B_1, \dots, B_k$, each consisting of at most $\log n$ consecutive locations of A . Each processor P_i determines the number d_i of locations in B_i that were not cancelled. The numbers $D_i = \sum_{j < i} d_j$ are computed. Similar considerations apply for the deletion algorithm.

Next, each processor P_i determines the set ρ_i of indices $< N-M$ that were cancelled by processor P_i and its cardinality r_i . The numbers $R_i = \sum_{j < i} r_j$ are computed. Each processor P_i writes the indices in ρ_i in places $R_i+1, \dots, R_i+r_i$ of some vector C . Once all processors are done with this, P_i copies the locations of block B_i that were not cancelled in those locations of A whose indices are in places $D_i+1, \dots, D_i+d_i$ of vector C .

Later in the split algorithm, every processor may create $O(\log n)$ new nodes. Storage allocation is handled as in the case of insertions. This applies for the deletion and union algorithms, as well.

Acknowledgments

The authors thank Professor J. Berstel for inspiring discussions.

REFERENCES

- [AHU] A. Aho, J. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley, Reading, Massachusetts, 1974.
- [AKS] M. Ajtai, J. Komlos and E. Szemerdi, "An $O(n \log n)$ sorting network", Proc. Fifteenth ACM Symp. on Theory of Computing, 1983, to appear.

- [BS] R. Bayer and M. Schkolnick, "Concurrency of operations on B-Trees",
Acta Informatica 9, 1-21 (1977).
- [FW] S. Fortune and J. Wyllie, "Parallelism in random access machines",
Proc. Tenth ACM Symp. on Theory of Computing, 114-118 (1978).
- [S] Ellis C. Schlatter, "Concurrent search and insertion in 2-3 trees",
Acta Informatica 14, 63-86 (1980).
- [V1] U. Vishkin, "Implementation of simultaneous memory access in models
that forbid it", TR 210, Dept. of Computer Science, Technion,
Haifa, Israel (1981), and J. of Algorithms 4, 45-50, (1983).
- [V2] U. Vishkin, "Parallel-Design space Distributed-Implementation
space (PDDI) general purpose computer", RC 9541, IBM T.J.
Watson Research Center, Yorktown Heights, N.Y. 10598,
1982.
- [W] H. Wagener, "Parallele Bearbeitung von 2-3-Bäumen", Diplomarbeit,
Fakultät für Mathematik, Universität Bielefeld, 1982.

A fine will be charged for each day the book is kept overtime.

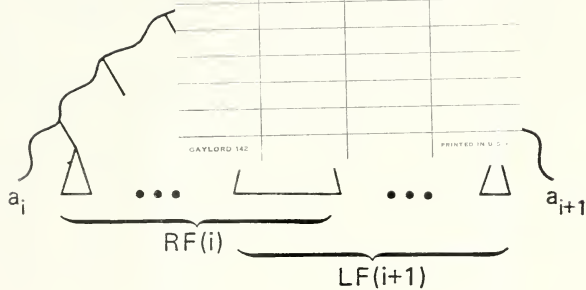


FIGURE 1.

NYU
Comp. Sci. Dept.
TR-70
Paul
Parallel computation on

c.1

NYU
Comp. Sci. Dept.
TR-70 Paul

c.1

AUTHOR
Parallel computation on

TITLE
2-3 trees.

DATE DUE	BORROWER'S NAME

LIBRARY
N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012

